

# 3D Rendering with Ray Tracing in Python

Calvin Godfrey

Virginia Polytechnic Institute and State University

25 February 2022

# Table of Contents

- 1 About Me
- 2 Intro to Computer Graphics
- 3 Provided Code
- 4 Implementing new Code
- 5 Shapes and Intersections
- 6 Materials
- 7 Suggested Improvements and Resources

# About Me

- Senior at Virginia Tech
- Began ray tracing in April 2020



*Rendered in June 2020; 11 hours*



*Rendered in June 2020; 8.5 hours*

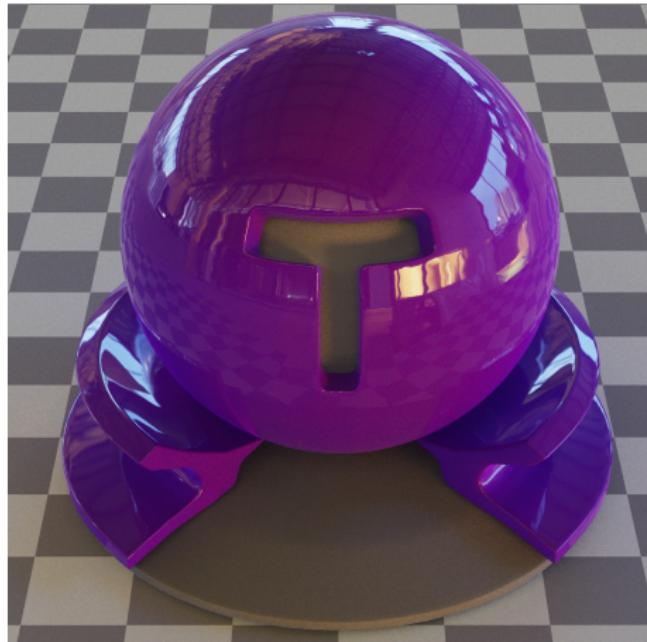
# About Me

- Senior at Virginia Tech
- Began ray tracing in April 2020



*Rendered in December 2020; 2.5 hours*

Calvin Godfrey

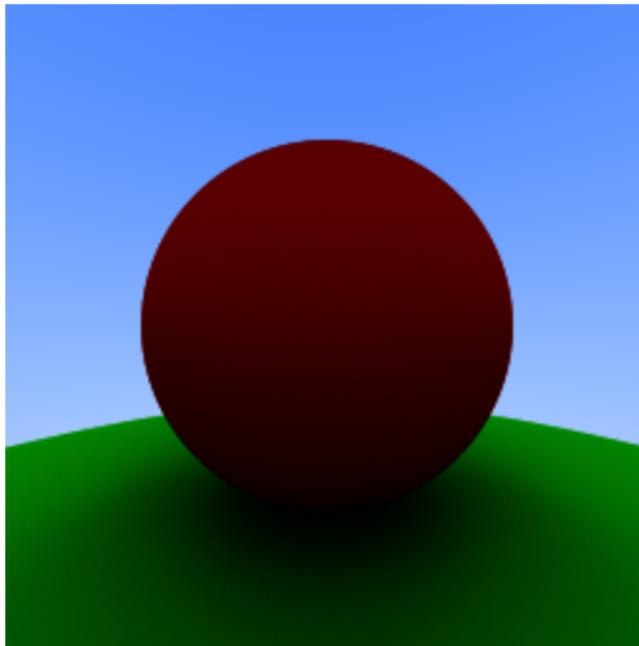


*Rendered in January 2021; 3.5 hours*

3 / 48

# Goal of the Talk

- Condense my first few weeks of learning into an hour
- Provide a flexible code base to continue developing new features
- Final image generated by this workshop is shown below

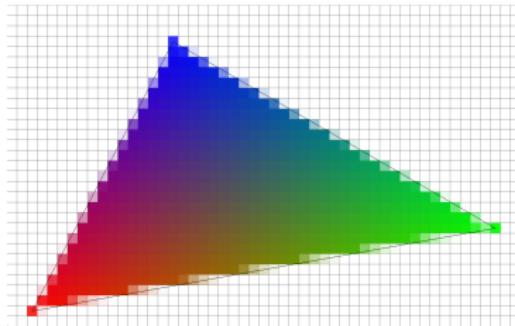


# Table of Contents

- 1 About Me
- 2 Intro to Computer Graphics
- 3 Provided Code
- 4 Implementing new Code
- 5 Shapes and Intersections
- 6 Materials
- 7 Suggested Improvements and Resources

# Computer Graphics

- Software Rasterization

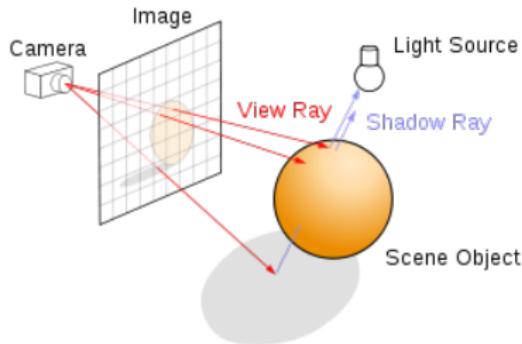


- “Given *shape*, what pixels should show this?”
- Faster, commonly used in cartoon-styled games
- Not the focus of this workshop

- Ray Tracing

# Computer Graphics

- Software Rasterization
- Ray Tracing



- “Given *pixel*, what object should this show?”
- Slower, but more realistic
- Involves the study of physics (light interactions), math (sampling techniques) and computer science (acceleration structures)

# Table of Contents

- 1 About Me
- 2 Intro to Computer Graphics
- 3 Provided Code**
- 4 Implementing new Code
- 5 Shapes and Intersections
- 6 Materials
- 7 Suggested Improvements and Resources

# Vector

```
class Vector3:  
    def __init__(self, x, y, z):  
        self.x, self.y, self.z = x, y, z
```

- Standard vector operations; addition and scalar multiplication
- Also contains utility functions for dot/cross product, length, normalize, etc.
- Used to represent points, vectors, and colors

# Ray

```
class Ray:  
    def __init__(self, origin, direction):  
        self.origin = origin  
        self.direction = direction  
  
    def at(self, t):  
        return self.orgin + t * self.direction
```

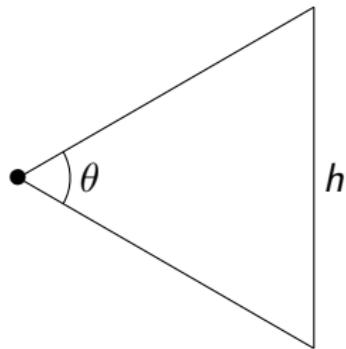
- Contains origin of ray and direction (both Vector3)
- Includes method to find point at given distance on ray

# ImageWrapper

```
class ImageWrapper:  
    def __init__(self, name, width, height):  
        # ...  
  
    def save(self):  
        # ...  
  
    def write_pixel(self, x, y, color):  
        # Assumes color is Vector3 and  
        # 0 <= x < width, 0 <= y < height  
        # ...
```

- Constructor takes file name and file size
- Provides method to save file and write pixel at specified coordinate

# Camera



- Requires location, view direction (target), and orientation ('up' direction), field of view, and aspect ratio
- Field of view ( $\theta$ ) controls 'zoom' of image; higher field of view is more zoomed out

# Camera

- Constructor takes all that information and stores information about the projected plane
- Contains an additional ray generation method that will be useful later

```
class Camera:  
    def __init__(self, loc, to, up, aspect_ratio, vfov):  
        """Parameters:  
            location      -- Vector3  
            direction     -- Vector3 target of camera  
            up            -- 'up' direction relative to the camera.  
            aspect_ratio   -- width / height of plane  
            vfov          -- vertical field of view, in degrees"""  
        # implementation not included  
  
    def generate_ray(self, x, y):  
        """Takes an (x, y) pair (in [0, 1]), returns  
        ray that starts at the camera that goes  
        through that point on the plane."""  
        # implementation not included
```

# Table of Contents

- 1 About Me
- 2 Intro to Computer Graphics
- 3 Provided Code
- 4 Implementing new Code
- 5 Shapes and Intersections
- 6 Materials
- 7 Suggested Improvements and Resources

# Main Rendering Loop

- First, define constants, Camera, and ImageWrapper

```
def main():
    # Define constants, ImageWrapper, and Camera
    aspect_ratio = 1
    height = 256
    width = int(height * aspect_ratio) # force to be integer
    name = "test.png"
    wrapper = ImageWrapper(name, width, height)
    # Camera pointing from origin to (0, 1, 0) with z as up
    camera = Camera(Vector3(0, 0, 0), Vector3(0, 1, 0),
                     Vector3(0, 0, 1), aspect_ratio, 90)
    #
    #
    #
    #
    #
    #
    # ...
```

# Main Rendering Loop

- Then iterate over each pixel and calculate color

```
def main():
    # Define constants, ImageWrapper, and Camera
    aspect_ratio = 1
    height = 256
    width = int(height * aspect_ratio) # force to be integer
    name = "test.png"
    wrapper = ImageWrapper(name, width, height)
    # Camera pointing from origin to (0, 1, 0) with z as up
    camera = Camera(Vector3(0, 0, 0), Vector3(0, 1, 0),
                     Vector3(0, 0, 1), aspect_ratio, 90)
    for y in range(height):
        for x in range(width):
            ray = camera.generate_ray(x / width, y / height)
            color = # ... some calculation to generate color
            wrapper.write_pixel(x, y, color)
    wrapper.save()
```

# First Render Test

- Set color of pixel depending on  $z$  component of ray's direction

```
# ...
white = Vector3(1, 1, 1)
blue = Vector3(0.5, 0.7, 1)
for y in range(height):
    for x in range(width):
        ray = camera.generate_ray(x / width, y / height)
        # map z in [-1, 1] to [0, 1]
        t = (ray.direction.normalize().z + 1) / 2
        # Linearly interpolate between white and blue
        color = t * blue + (1 - t) * white
        wrapper.write_pixel(x, y, color)
wrapper.save()
```

# First Render Test

- Set color of pixel depending on  $z$  component of ray's direction

Resulting image:



# Table of Contents

- 1 About Me
- 2 Intro to Computer Graphics
- 3 Provided Code
- 4 Implementing new Code
- 5 Shapes and Intersections
- 6 Materials
- 7 Suggested Improvements and Resources

# Sphere-Ray Intersection

- Given ray with origin  $O$  and direction  $D$ , where does it first intersect a sphere with center  $C$  and radius  $R$ ?
- Equivalent question is intersection between ray at  $O - C$  and direction  $D$  and sphere centered at origin with radius  $R$ 
  - Parameterize ray as  $(O - C) + tD$ ; each value of  $t$  gives new point  $P$  on ray
  - $P$  is on sphere if  $|P|^2 = R^2$ , or  $|P|^2 - R^2 = 0$ .
  - But  $P = (O - C) + tD$ , giving the following equation:

$$|(O - C) + tD|^2 - R^2 = 0$$

# Sphere-Ray Intersection

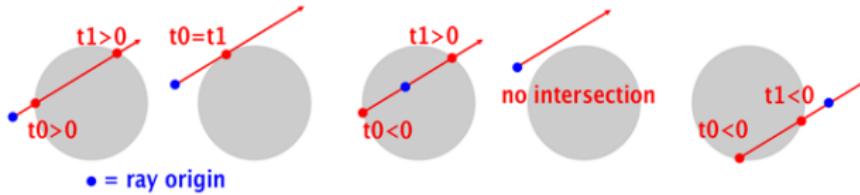
$$\begin{aligned}|(O - C) + tD|^2 = R^2 &\iff \\((O - C) + tD) \cdot ((O - C) + tD) = R^2 &\iff \\O \cdot O - 2O \cdot C + 2t(O \cdot D) + C \cdot C - 2t(C \cdot D) + t^2(D \cdot D) = R^2 &\iff \\(D \cdot D)t^2 + (2O \cdot D - 2C \cdot D)t + O \cdot O + C \cdot C - 2O \cdot C = R^2 &\iff \\(D \cdot D)t^2 + 2D \cdot (O - C)t + (O - C) \cdot (O - C) - R^2 = 0\end{aligned}$$

where  $\cdot$  indicates the dot product between two vectors

- This is a quadratic equation in  $t$  with  $a = D \cdot D$ ,  $b = 2D \cdot (O - C)$ , and  $c = (O - C) \cdot (O - C) - R^2$
- Two solutions to quadratic formula give two (possible) intersection points

# Geometric Interpretation of Ray-Sphere Formula

- Two real solutions: Ray intersects with sphere twice
  - Two positive solutions: Two intersections “in front” of ray (example 1 below)
  - One positive, one negative solution: Ray starts within sphere (example 3)
  - Two negative solutions: Sphere is “behind” ray (example 5)
- One solution: Ray “grazes” sphere at single point (example 2)
- No real solution: Ray misses sphere (example 4)



# Quadratic Formula Solver

```
import math
def solve_quadratic(a, b, c):
    desc = b * b - 4 * a * c
    if desc < 0: # No real solution
        return []
    if desc == 0: # Single solution
        return [-b / (2 * a)]
    else:
        sqrt_desc = math.sqrt(desc)
        b_minus = (-b - sqrt_desc) / (2 * a)
        b_plus = (-b + sqrt_desc) / (2 * a)
        # Always return smaller solution first
        return [min(b_minus, b_plus), max(b_minus, b_plus)]
```

# Sphere-Ray Intersection

```
def sphere_intersection(center, radius, ray):
    diff = ray.origin - center #  $O - C$ 
    a = ray.direction.dot(ray.direction) #  $D \cdot D$ 
    b = diff.dot(2 * ray.direction) #  $2D \cdot (O - C)$ 
    c = diff.dot(diff) - radius * radius
    solutions = solve_quadratic(a, b, c)
    # Because solutions are in increasing order,
    # we can iterate until we find a positive solution
    for solution in solutions:
        if solution > 0:
            return True
    return False
```

# Rendering a Sphere

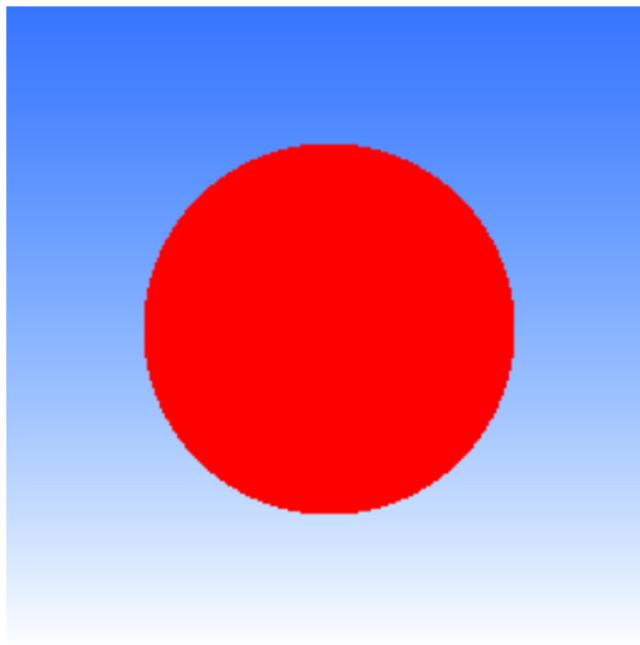
- Now update the render loop from earlier to use this

```
# ...
white = Vector3(1, 1, 1)
blue = Vector3(0.5, 0.7, 1)
for y in range(height):
    for x in range(width):
        ray = camera.generate_ray(x / width, y / height)
        center = Vector3(0, 10, 0)
        radius = 5
        if sphere_intersection(center, radius, ray):
            color = Vector3(1, 0, 0) # red
        else: # background color
            t = (ray.direction.normalize().z + 1) / 2
            color = t * blue + (1 - t) * white
        wrapper.write_pixel(x, y, color)
wrapper.save()
```

# Rendering a Sphere

- Now update the render loop from earlier to use this

And the new result:



# Making a Sphere Class

- Sphere contains center, radius, color, and intersection method

```
class Sphere:  
    def __init__(self, center, radius, color):  
        self.center = center  
        self.radius = radius  
        self.color = color  
  
    def intersection(self, ray):  
        diff = ray.origin - self.center  
        a = ray.direction.dot(ray.direction)  
        b = diff.dot(2 * ray.direction)  
        c = diff.dot(diff) - self.radius * self.radius  
        solutions = solve_quadratic(a, b, c)  
        for solution in solutions:  
            if solution > 0:  
                return (True, self.color)  
        return (False, self.color)
```

# Storing Intersection Information

- Often require knowing a lot of information about intersection; for now we just have location and color, but in more robust renderers, it can store texture coordinates, information about the material, etc.
- Store the information in a HitRecord object

```
class HitRecord:  
    def __init__(self, point, color, time):  
        self.point = point  
        self.color = color  
        self.time = time # time along ray
```

# Updated Sphere Intersection

- We can use the new HitRecord in the Sphere intersection method

```
class Sphere:  
    # ...  
    def intersection(self, ray):  
        diff = ray.origin - self.center  
        a = ray.direction.dot(ray.direction)  
        b = diff.dot(2 * ray.direction)  
        c = diff.dot(diff) - self.radius * self.radius  
        solutions = solve_quadratic(a, b, c)  
        for solution in solutions:  
            if solution > 0: # new code to make HitRecord  
                p = ray.at(solution)  
                return HitRecord(p, self.color, solution)  
    return None
```

# Rendering Multiple Spheres

- Go through all spheres, find the one with smallest positive intersection time
- Return color associated with it

```
# takes ray, list of spheres; returns color
def get_intersection(ray, spheres):
    nearest = 1e100 # big number
    found_record = None
    for sphere in spheres:
        record = sphere.intersection(ray)
        if record is None:
            continue
    # ...
    #
    #
    #
    #
    #
    #
    #
```

# Rendering Multiple Spheres

- Go through all spheres, find the one with smallest positive intersection time
- Return color associated with it

```
# takes ray, list of spheres; returns color
def get_intersection(ray, spheres):
    nearest = 1e100 # big number
    found_record = None
    for sphere in spheres:
        record = sphere.intersection(ray)
        if record is None:
            continue
        if record.time < nearest:
            found_record = record
            nearest = record.time
    if found_record == None:
        # This does simple gradient from earlier
        return get_background(ray)
    return found_record.color
```

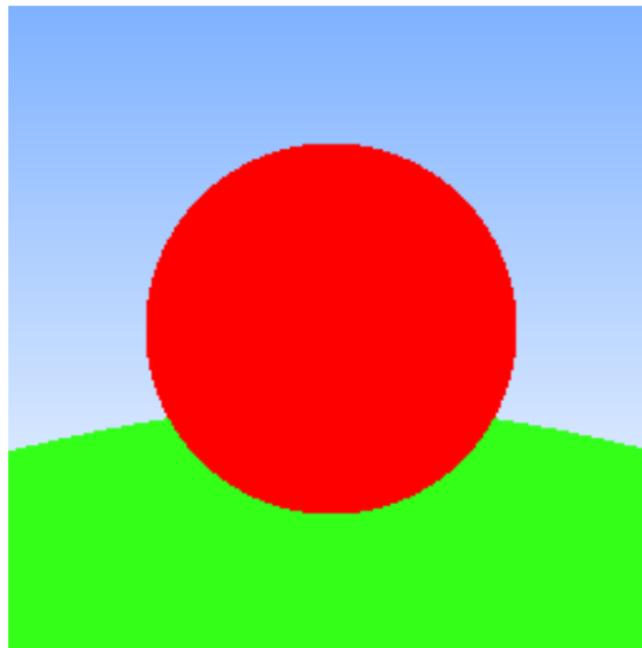
# Updating Scene

- Make list of all the spheres in the scene

```
# ...
# same constants as before
spheres = [Sphere(Vector3(0, 10, 0), 5, Vector3(1, 0, 0)),
           Sphere(Vector3(0, 10, -100), 95, Vector3(0.2, 1, 0.1))]
for y in range(height):
    for x in range(width):
        ray = camera.generate_ray(x / width, y / height)
        color = get_intersection(ray, spheres)
        wrapper.write_pixel(x, y, color)
wrapper.save()
```

# Updating Scene

- Make list of all the spheres in the scene



# Table of Contents

- 1 About Me
- 2 Intro to Computer Graphics
- 3 Provided Code
- 4 Implementing new Code
- 5 Shapes and Intersections
- 6 Materials
- 7 Suggested Improvements and Resources

# Shading

- So far, everything looks flat. How can we give the illusion of depth?
- Right now, light stops once it hits something, which isn't very realistic
- In real life, light bounces around until it is absorbed by something
- Even though our rays start at the camera, the same effect occurs
- Changing the way rays of light bounce changes how the material looks

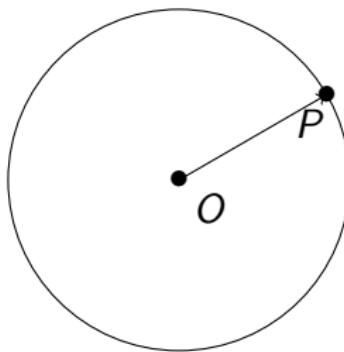
# Lambertian Material

- Easiest material to simulate is called "lambertian"; models rough surfaces, looks the same from all angles
- Need to generate new direction for a ray, for lambertian material this is done with generating an arbitrary unit vector

```
def rand_on_sphere():
    # random.random() returns [0, 1),
    # map to the range [-1, 1]
    v = Vector3(2 * random.random() - 1,
                2 * random.random() - 1,
                2 * random.random() - 1)
    return v.normalize() # make unit vector
```

# Calculate Normal of Sphere

- Normal is unit vector perpendicular to the surface
- Used in calculate how a ray bounces
- For sphere, normal is vector from center to point



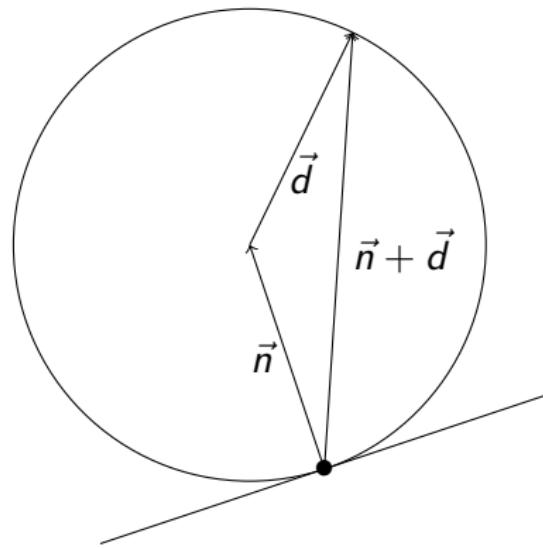
# Calculate Normal of Sphere

- Normal is unit vector perpendicular to the surface
- Used in calculate how a ray bounces
- For sphere, normal is vector from center to point

```
def sphere_normal(self, p):  
    return (p - self.center).normalize()
```

# Generating new Ray Direction

- Given normal  $\vec{n}$  and randomly generated unit vector  $\vec{d}$ , the new direction is  $\vec{n} + \vec{d}$



# Updating HitRecord and Sphere Intersection

- We must first update HitRecord to store the normal vector
- Then update sphere intersection to calculate this information

```
# ...
# rest is the same
for solution in solutions:
    if solution > 0: # new code to make HitRecord
        p = ray.at(solution)
        normal = (p - self.center).normalize()
        # add normal as parameter to HitRecord
        return HitRecord(p, self.color, solution, normal)
```

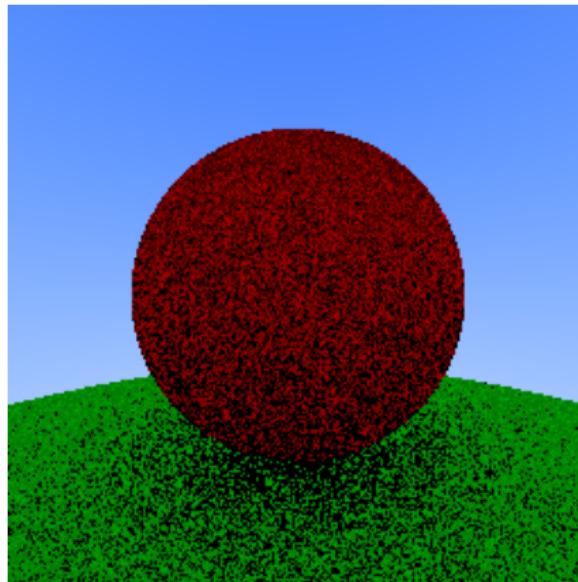
## Updating get\_intersection

- We must now store the depth of the traversal, how many bounces we've done, and eventually stop; call with arbitrary depth (like 20)

```
def get_intersection(ray, spheres, depth):  
    if depth == 0: # too many bounces  
        return Vector3(0, 0, 0) # black  
    found_record = None  
    # same logic to find closest intersection as before  
    # ...  
    # generate new ray direction  
    direction = found_record.normal + rand_on_sphere()  
    new_ray = Ray(found_record.point, direction)  
    # color of next bounce  
    next_color = get_intersection(new_ray, spheres, depth - 1)  
    current_color = found_record.color  
    # This is component-wise multiplication  
    return current_color.multiply(next_color)
```

## Render with Shadows

- Now the image has a lot of noise
- A single ray can't capture enough information to accurately estimate what the image should look like
- Increase accuracy by sending multiple rays per pixel



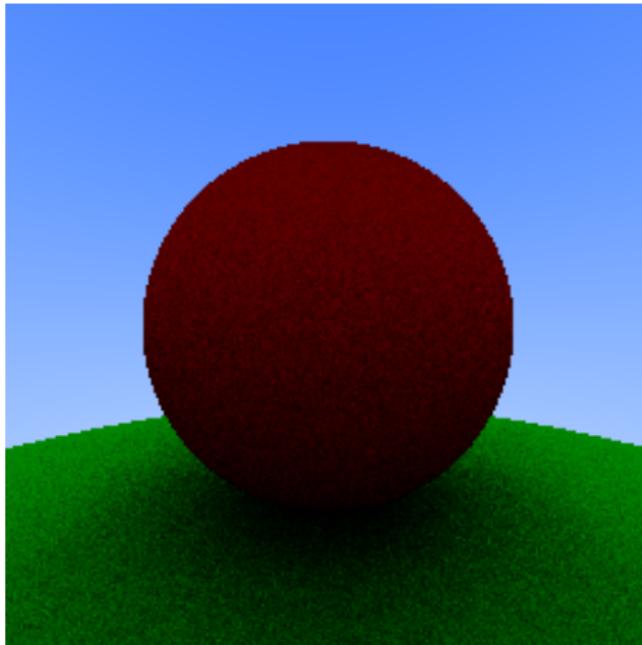
# Multiple Samples

- Set color to average of all samples
- Taking  $n$  samples per pixel makes render take  $n$  times longer to finish

```
# ...
samples = 50
for y in range(height):
    for x in range(width):
        final_color = Vector3(0, 0, 0)
        for _ in range(samples):
            ray = camera.generate_ray(x / width, y / height)
            color = get_intersection(ray, spheres, 20)
            final_color += color
        final_color /= samples
        wrapper.write_pixel(x, y, final_color)
wrapper.save() # save more often
```

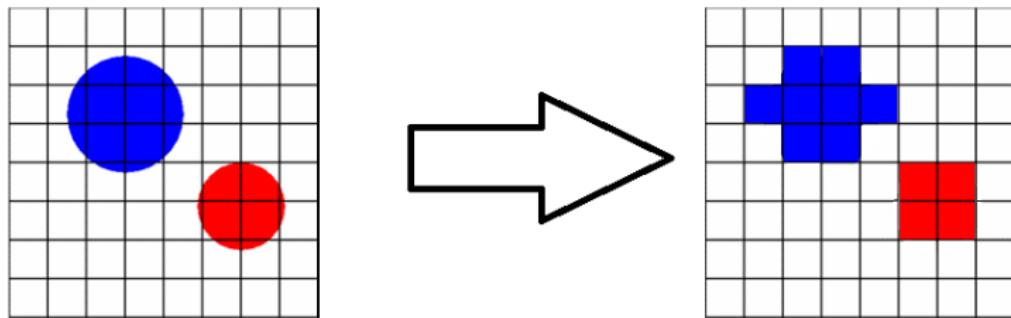
# Multiple Samples

- Set color to average of all samples
- Taking  $n$  samples per pixel makes render take  $n$  times longer to finish



# Anti-Aliasing

- Image still contains harsh boundaries between sphere and background
- Each pixel only contains what's at the center of the pixel



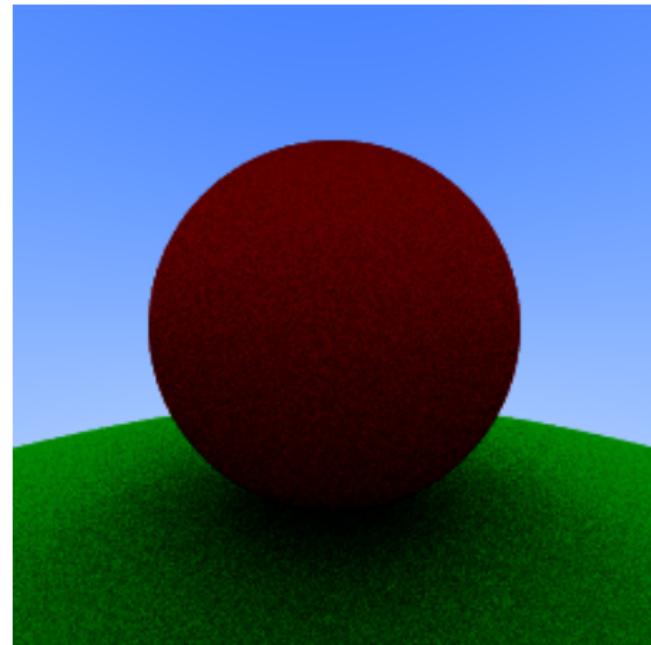
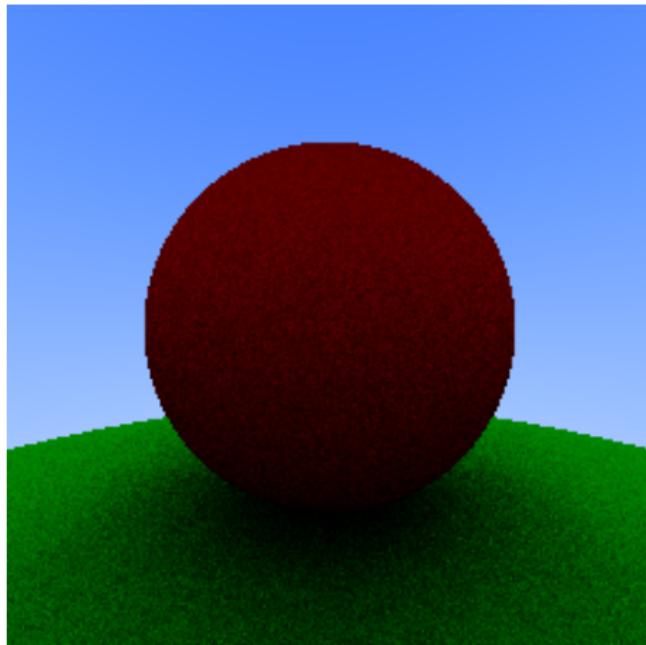
# Anti-Aliasing

- Nudging each original ray better represents what the pixel shows

```
samples = 50
for y in range(height):
    for x in range(width):
        final_color = Vector3(0, 0, 0)
        for sample in range(samples):
            ox = x + random.random() # offset x
            oy = y + random.random() # offset y
            ray = camera.generate_ray(ox / width, oy / height)
            color = get_intersection(ray, spheres, MAX_DEPTH)
            final_color += color
        final_color /= samples
        wrapper.write_pixel(x, y, final_color)
wrapper.save()
```

# Anti-Aliasing

- Nudging each original ray better represents what the pixel shows



# Lambertian Class

- We want to eventually have multiple materials, so lets abstract the material into a class
- Stores the color and has single method to return new direction for ray

```
class Lambertian:  
    def __init__(self, color):  
        # color is Vector3  
        self.color = color  
  
    def scatter(self, record, ray):  
        # returns direction for next ray  
        return record.normal + rand_on_sphere()
```

# Updating Sphere Class

- Sphere class now has a material member instead of color

```
class Sphere:  
    def __init__(self, center, radius, material):  
        self.center = center  
        self.radius = radius  
        self.material = material # updated  
    def intersection(self, ray):  
        diff = ray.origin - self.center  
        a = ray.direction.dot(ray.direction)  
        b = diff.dot(2 * ray.direction)  
        c = diff.dot(diff) - self.radius * self.radius  
        solutions = solve_quadratic(a, b, c)  
        for solution in solutions:  
            if solution > 0:  
                p = ray.at(solution)  
                normal = (p - self.center).normalize()  
                # Now we also have to update HitRecord  
                return HitRecord(p, self.material, solution, normal)  
    return None
```

# Updating HitRecord Class

- Similarly, HitRecord now has a material member instead of color

```
class HitRecord:  
    def __init__(self, point, material, time, normal):  
        self.point = point  
        self.normal = normal  
        self.time = time  
        self.material = material
```

# Updating get\_intersection Method

- Use scatter on the material in the HitRecord to get the new direction

```
# generate new ray direction
direction = found_record.material.scatter(found_record, ray)
new_ray = Ray(found_record.point, direction)
curr_color = found_record.material.color
next_color = get_intersection(new_ray, sphere, depth - 1)
return curr_color.multiply(next_color)
```

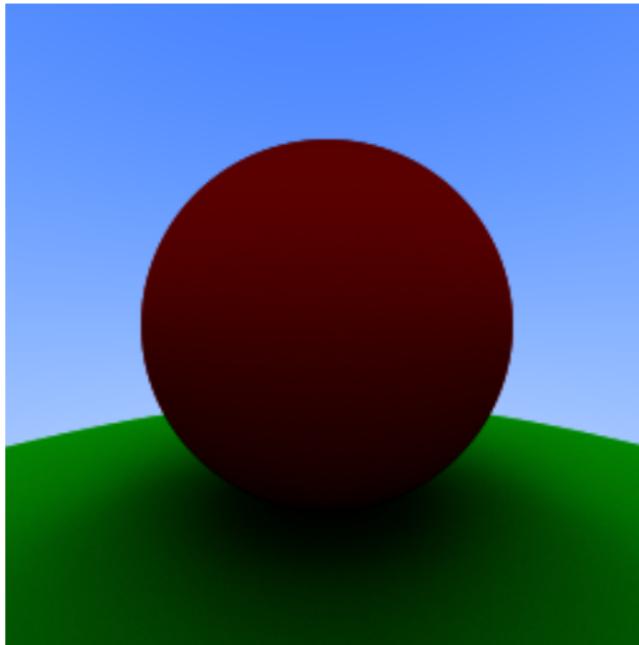
# Updating Render Loop

- Construct spheres using new Lambertian class

```
red = Lambertian(Vector3(1, 0, 0))
green = Lambertian(Vector3(0.2, 1, 0.1))
spheres = [Sphere(Vector3(0, 10, 0), 5, red),
           Sphere(Vector3(0, 10, -100), 95, green)]
# same loop body as before
```

# Updating Render Loop

- Resulting image is the same, but the code is more flexible
- This is the final scene rendered at 5,000 samples per pixel



# Table of Contents

- 1 About Me
- 2 Intro to Computer Graphics
- 3 Provided Code
- 4 Implementing new Code
- 5 Shapes and Intersections
- 6 Materials
- 7 Suggested Improvements and Resources

## What next?

- Implement reflective, metallic material (reflect ray about normal vector)
- Implement transparent glass-like material (refract ray using Snell's law)
- Modify the camera and use the `focal_length` variable to create a depth-of-field effect
- Add texture coordinates so that you can wrap an image on a sphere
- Add support for more shapes
  - If you add support for triangles, you can import meshes (.obj files) and render arbitrary objects
- Make it multi-threaded (rendering is “embarrassingly parallelizable”)

# Further Reading and Resources

- Ray Tracing in a Weekend (<https://raytracing.github.io/>)
  - These slides are *heavily* inspired by this series
- Physically Based Rendering (<http://www.pbr-book.org/3ed-2018/contents.html>)
  - Very in-depth (sometimes too in-depth); also very advanced and dense
- Advanced Global Illumination, 2nd Edition
  - Math-heavy, but also thorough (I'm currently reading this)
- An in-depth look at a variety of materials ([https://blog.selfshadow.com/publications/s2013-shading-course/hoffman/s2013\\_pbs\\_physics\\_math\\_slides.pdf](https://blog.selfshadow.com/publications/s2013-shading-course/hoffman/s2013_pbs_physics_math_slides.pdf))
- Discord server about 3D graphics (<https://discord.gg/6mgNGk7>)
- List of websites with meshes that you can render:
  - <https://casual-effects.com/data/>
  - <https://threedescans.com/>
  - <https://pbrt.org/scenes-v3.html>